Big-O Notation Explained

This Post will focus on simplified explanation of the Big-O Notation. in this blog I will explain to you about what is Big-O Notation, how important it is related to Algorithms. Before I start to tell you about this magic, let me give you some quotes.

The art of programming is the art of organizing complexity.

Edsger W. Dijkstra

…

What is Algorithm?

~~Algorithm is a word used by programmers when they don’t want to explain what they did.❌~~

Algorithm is a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.✅

We can say that algorithm is a series of a processes executed in sequence to break a problem. Algorithm can be vary depend on who made it, but the question is, which algorithm has the least complexity?

We are facing so many problem on our daily life, right? for example, We want to go to our workplace, we know that there are various route that we can go through, but the question is, which route is the most effective and efficient for us? we can use time complexity analysis in order to find the best route for us.

…

What is Big-O Notation?

Before we talk about this very particular notation, let me explain to you about algorithm complexity. Mainly, there are two types of complexity :

1. Space Complexity
2. Time complexity

Space complexity is how much memory does the algorithm need to do a process. While time complexity is how long does it take for an algorithm to do a process. From those two types of complexity, we can break it down to a diagram below.

Big-O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big-O is the method of expressing the upper bound of an algorithm’s running time, It is measure the longest amount of time it could possibly take for the algorithm to complete (Patel, 2018). we can simplify Patel’s explanation about Big-O with this :

“Big-O describes the worst-case scenario of algorithm’s running time”

Categories of Big-O notation

O(1) -> constant computing time

O(n) -> linear computing time

O(n2) -> quadratic computing time

O(2n) -> exponential computing time

O(log n) -> logarithmic computing time

|  |  |
| --- | --- |
| Regular notation | Big-O Notation |
| 4 | O(1) |
| 4n+90 | O(n) |
| 12n2 | O(n2) |

“In Big-O, the notation ignore constant with smaller impact to the processes”

Table showing rate of growth of function

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | Log N | N log N | N2 | N3 | 2N |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 10 | 1 | 10 | 100 | 1000 | 1024 |
| 50 | 7 | 350 | 2500 | 125000 | 1020 |
| 100 | 10 | 1000 | 10000 | 1000000 | 1030 |

From the table above, we can see that if algorithm takes time O(log n), it is faster than O(n) for large amount of data. The same thing happened with O(n log n), it is faster than O(n2) but not as fast as O(n)

1. O(1)

O(1) describes that an algorithm will always run in O(1) time (“constant”) regardless of the amount of input data. The input could be just 1 item or 1,000,000 items, but this algorithm would still require a single “step”.

example :

def isEmpty(list):

return list[0]==null

the example above shows that the input list could be just 1 item or 1,000,000 items, but the function just require just one “step”.

1. O(n)

O(n) describes that an algorithm’s running time will grow linearly as the size of the input data set is increasing.

example :

def printArray(List):

for i in range(len(List)):

print(List[i])

printArray function shows that runtime of the function grow linearly. This means if the array has 100 items, I have to print 100 times. if it has 100,000 items, I have to print 100,000 times.

1. O(n2)

O(n2) describes that an algorithm’s running time will grow proportional to the square of the size of the input data set. This type of process is common with algorithm that involve nested loop over the data set. The deeper the nest will affect the runtime and resulting O(n3), O(n4), and so on.

example :

def printMatrix(List):

for I in range(len(List):

for j in range(len(List[0])):

print(List[i][j])

Here we have two nested loop. If we have a matrix with the size n\*n, the outer loop will run n times and the inner loop will also run n times. Thus this function runs in O(n2) or quadratic time.

explanation : O(n)\*O(n) = O(n2)

1. O(2n)

O(2n) describes an algorithm’s running time whose growth doubles with each addition to the input data set. The growth of this function is exponential-starting very shallow, then raised significantly. Recursive algorithm that solve a problem of size N by recursively solving two smaller problems of size n-1 are often have O(2n) running time.

example :

def Fibonacci(n):

if (n<=1):

return n

return Fibonacci(n-2) + Fibonacci(n-1)

1. O(log n)

An algorithm with O(log n) running time often has these two attributes:

* The choice of the next element on which to perform some action is one of several possibilities
* Only one will need to be chosen

or

* The elements on which the action is performed are digits of n

This is why looking up someone’s phone number is O(log n). We don’t need to check every single person to find the right person; instead, we can simply divide-and-conquer by lookin based on where their name is alphabetically, and in every section you only need to explore a subset of each section to find someone’s phone number.

1. Drop the less significant term

def printAllNumbersThenAllPairSums(List,size):

for i in range(size):

print(List[i])

for i in range(size):

for j in range(size):

print(List[i]+List[j])

Here is our runtime :

O(n) + O(n\*n) = O(n+n2), which we just call it O(n2)

SUMMARY

So, by understanding The Big-O Notation, it will be easier for us to know which algorithm is the most efficient for the problem we are facing.